FORMAL ANALYSIS OF SUZUKI&KASAMI DISTRIBUTED MUTUAL EXCLUSION ALGORITHM

Kazuhiro Ogata*
Graduate School of Information Science, JAIST
ogata@jaist.ac.jp

Kokichi Futatsugi
Graduate School of Information Science, JAIST
kokichi@jaist.ac.jp

Abstract Since parallel and distributed algorithms are subject to subtle errors that are unlikely to be detected in usual operation, only testing is not enough to reduce errors. Thus, it is necessary to formally analyze such algorithms in order to confirm that they have desirable properties. This paper describes the case study that Suzuki&Kasami distributed mutual exclusion algorithm is formally analyzed. In the case study, the algorithm has been modeled using UNITY-like models called observational transition systems (OTS's), the model has been described in CafeOBJ, and it has been verified that the algorithm is mutually exclusive and lockout free with the help of CafeOBJ system. In the verification that the algorithm is lockout free, we have found a hidden assumption necessary for the verification, which is not explicitly mentioned in the original paper written by Suzuki and Kasami.

Keywords: CafeOBJ, parallel and distributed algorithms, modeling, observational transition systems (OTS's), UNITY, verification

1. Introduction

Parallel and distributed algorithms are subject to subtle errors that are unlikely to be detected in usual operation. Moreover, they are inherently nondeterministic. Therefore, only testing is not enough to reduce errors. It is

*The current affiliation is SRA Key-Technology Laboratory.

necessary to formally analyze such algorithms in order to confirm that they have desirable properties and to reduce errors.

The way, used in this case study, to analyze parallel and distributed algorithms is that the algorithms are formally modeled, the models are described in a specification language, and it is verified that the algorithms have some desirable properties based on the specifications with the help of an interactive theorem prover. Observational transition systems, or OTS's, which are reformulation of UNITY (Chandy and Misra, 1988) models, are used to model the algorithms, and CafeOBJ (CafeOBJ web page, 2001; Diaconescu and Futatsugi, 1998), an algebraic specification language, is used to describe the models. Since CafeOBJ system includes some verification tools, it is used as an interactive theorem prover. The algorithm analyzed in this case study is Suzuki&Kasami distributed mutual exclusion algorithm (Suzuki and Kasami, 1985).

In the case study, it has been verified that Suzuki&Kasami algorithm is mutually exclusive and lockout free. In the verification that the algorithm is lockout free, we have found a hidden assumption necessary for the verification, which is not explicitly mentioned in the original paper (Suzuki and Kasami, 1985) written by Suzuki and Kasami.

2. Basic Computational Models: Transition Systems

UNITY (Chandy and Misra, 1988) models are reformulated in the same manner as the definition of fair transition systems (Manna and Pnueli, 1991; Manna and Pnueli, 1995), which are called observational transition systems, or OTS's. We can use an OTS to model a parallel and distributed algorithm. An OTS $\mathcal{S} = \langle \mathcal{V}, \mathcal{I}, \mathcal{T} \rangle$ consists of:

- $\mathcal{V}$: A set of typed variables. Each variable has its own type. The variables (or their possible values) form the state space Σ of $\mathcal{S}$, and a state of $\mathcal{S}$ is a point, or an element of Σ.

- $\mathcal{I}$: The initial condition. This condition specifies the initial values of the variables. Since some variables may not be specified by $\mathcal{I}$, $\mathcal{S}$ may have more than one initial state.

- $\mathcal{T}$: A set of transition rules. Each transition rule $\tau \in \mathcal{T}$ is a function $\tau : \Sigma \rightarrow \Sigma$ mapping each state $s \in \Sigma$ into a successor state $\tau(s) \in \Sigma$. Transition rules are generally defined together with conditions on which the transition rules are *effectively* executed, namely that their execution can change states of $\mathcal{S}$. If the condition of a transition rule is false in a state of $\mathcal{S}$, namely that the transition rule is not *effective* in the state, its execution does not change the state.

As defined, an OTS is deterministic with respect to each transition rule. That is, given a state $s \in \Sigma$ and a transition rule $\tau \in \mathcal{T}$, exactly one successor state $\tau(s) \in \Sigma$ is determined. Our purpose is not only description of an OTS as a model of a parallel and distributed algorithm in CafeOBJ but also verification that the parallel and distributed algorithm has some properties based on the CafeOBJ document with the help of CafeOBJ system. Hence, a deterministic transition system in this sense is more appropriate to CafeOBJ aided verification than a nondeterministic one because CafeOBJ system does not support nondeterministic execution, or term rewriting. Actually, such a deterministic transition system has practically enough power to model most parallel and distributed algorithms because if you need nondetermination with respect to a transition rule, you may achieve this by dividing the transition rule into multiple ones each of which is deterministic.

An execution starts from one initial state and goes on forever; in each step of execution some transition rule is selected nondeterministically and executed. Nondeterministic selection is constrained by the following fairness rule: every transition rule is selected infinitely often. Given an OTS, a set of infinite sequences of states is obtained from execution, constrained by the fairness rule, of the OTS. Such an infinite sequence of states is called a computation of the OTS. More specifically, a computation of an OTS $\mathcal{S}$ is an infinite sequence $s_0, s_1, \ldots$ of states satisfying:

- *Initiation* : For each $v \in \mathcal{V}$, v satisfies $\mathcal{I}$ in s_0.
- *Consecution* : For each $i \in \{0, 1, \ldots\}$, $s_{i+1} = \tau(s_i)$ for some $\tau \in \mathcal{T}$.
- *Fairness* : For each $\tau \in \mathcal{T}$, there exist an infinite number of indexes $i \in \{0, 1, \ldots\}$ such that $s_{i+1} = \tau(s_i)$.

A state of an OTS is called *reachable* if it appears in a computation of the OTS.

The concept *effectiveness* is similar to *enabledness* used in description of transition systems in temporal logic such as TLA (Lamport, 1994) or in a precondition-effect style such as I/O automata (Lynch, 1996).

When defining an OTS $\mathcal{S}$, another OTS $\mathcal{S}_{sub}$ may be used as a sub-system. The variables of $\mathcal{S}$ are the disjoint union of the variables explicitly defined in $\mathcal{S}$ and the variables defined in $\mathcal{S}_{sub}$. The initial condition specifies the initial values of all the variables. The condition may use the initial condition of $\mathcal{S}_{sub}$ for the variables defined in $\mathcal{S}_{sub}$. The transition rules are those explicitly defined in $\mathcal{S}$. But, we may connect a transition rule τ_{sub} defined in $\mathcal{S}_{sub}$ to a transition rule τ defined in $\mathcal{S}$ provided that some condition holds. If so, τ_{sub} is synchronously executed with τ when τ is executed in a state in which the condition holds.

3. Description of OTS's in CafeOBJ

CafeOBJ (CafeOBJ web page, 2001; Diaconescu and Futatsugi, 1998) is mainly based on two logical foundations: *initial* and *hidden* algebra. Initial algebra is used to specify abstract data types such as integers, and hidden algebra (Goguen and Malcolm, 2000) to specify objects in object-orientation. There are two kinds of sorts (corresponding to types in programming languages) in CafeOBJ. They are *visible* and *hidden* sorts. A visible sort represents an abstract data type, and a hidden sort the state space of an object. There are basically two kinds of operations to hidden sorts. They are *action* and *observation* operations, corresponding to so-called methods in object-orientation. An action operation, or an action can change a state of an object. It takes a state of an object and zero or more data, and returns another (possibly the same) state of the object. An observation operation, or an observation can be used to observe the value of a data component in an object. It takes a state of an object and zero or more data, and returns the value of a data component in the object. An action is basically specified with equations by describing how each observation changes relatively based on the values of observations in a state after executing the action in the state.

Declarations of visible sorts are enclosed with `[` and `]`, and those of hidden ones with `*[` and `]*`. Declarations of observations and actions start with `bop` or `bops`, and those of other operations with `op` or `ops`. After `bop` or `op` (or `bops` or `ops`), an operator is written (or more than one operator is written), followed by '`:`' and a sequence of sorts (i.e. sorts of the operators' arguments), and ended with '`->`' and one sort (i.e. sort of the operators' results). Definitions of equations start with `eq`, and those of conditional ones with `ceq`. After `eq`, two expressions, or terms connected by `=` are written, ended with a full stop. After `ceq`, two terms connected by `=` are written, followed by `if` and a term denoting a condition, and ended with a full stop.

Since objects can be regarded as transition systems, an OTS can be naturally described in CafeOBJ. The state space of an OTS is denoted by a hidden sort.

A single variable or a set of variables is represented by an observation. Given an OTS in which a set $\{x_i \mid i \in \{1,2,\ldots\}\}$ of variables which types are natural numbers is used, the variables are represented by an observation declared as `bop x : Sys NzNat -> Nat`, where `Sys` is the hidden sort denoting the state space of the OTS, and `NzNat` and `Nat` are the visible sorts denoting non-zero natural numbers and natural numbers respectively. The value of x_i in a state s is denoted by `x(`s`,`i`)`. The initial values of variables are specified with equations. An operation denoting any initial state is first declared as `op init : -> Sys`. Then if all x_i's are initially 0, we have the equation `eq x(init,I) = 0 .`, where `I` is a CafeOBJ variable whose sort is `NzNat`.

A single transition rule or a set of transition rules is represented by an action. Given an OTS in which a set $\{t_i \mid i \in \{1,2,\ldots\}\}$ of transition rules is used, the transition rules are represented by an action declared as `bop t : Sys NzNat -> Sys`. The successor state after executing t_i in a state s is denoted by `t(`s`,`i`)`. The behavior of transition rules is specified with equations that define how the state (i.e. the variables) changes if each transition rule is executed in a state. Suppose that if t_i is executed, x_i is incremented and x_j $(j \neq i)$ is left unchanged. Then, we have two equations:

```
eq x(t(S,I),I) = X(S,I) + 1 .
ceq x(t(S,I),J) = X(S,J) if I =/= J .
```

where `S`, `I` and `J` are CafeOBJ variables whose sorts are `Sys`, `NzNat` and `NzNat` respectively.

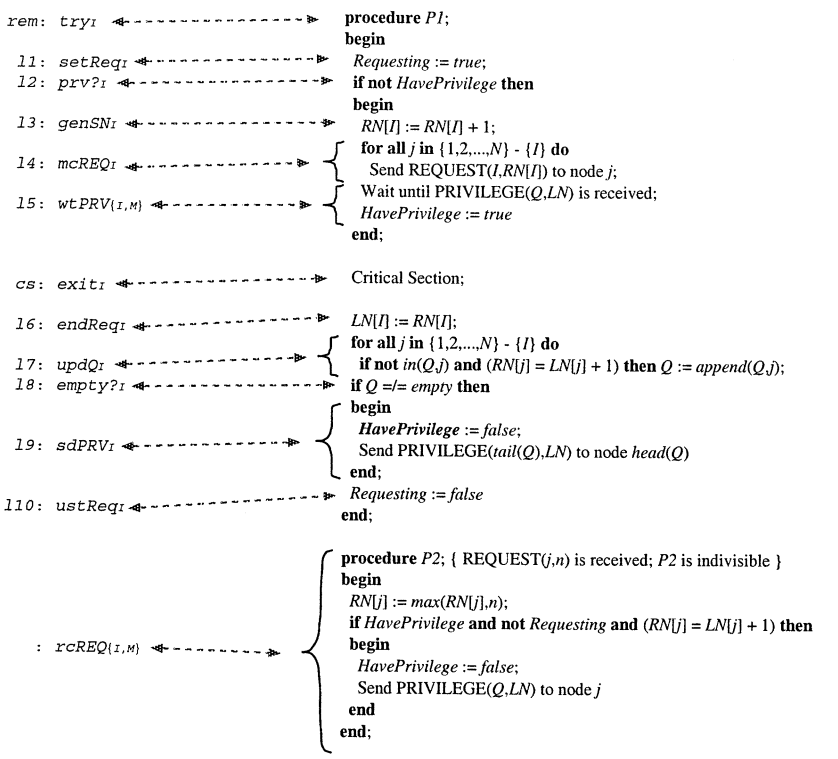
If an object represented by a hidden sort `S` has another object represented by a hidden sort $\mathtt{S}_{\mathtt{sub}}$ as a sub-system, or a component, then an operation from `S` to $\mathtt{S}_{\mathtt{sub}}$, called a *projection*, is used to represent the component (Diaconescu et al., 1999). Projections are also used to represent sub-systems of OTS's.

In description of an OTS in CafeOBJ, we first write the signature of the specification of the OTS, declaring sorts and operations, next write equations defining the initial values of the observations, and then write equations defining how a state of the OTC changes after each action is executed in that state.

4. Suzuki&Kasami Distributed Mutual Exclusion Algorithm

Let us consider a computer network consisting of a fixed number, say N (≥ 1), of nodes that have no memory in common and can communicate only by exchanging messages. The communication delay is totally unpredictable, namely that although messages eventually arrive at their destinations, they are not guaranteed to be delivered in the same order in which they are sent. The distributed mutual exclusion problem is to solve a mutual exclusion requirement for such a computer network, namely that at most one node may stay in its critical section at any moment. Suzuki and Kasami (Suzuki and Kasami, 1985) have presented a distributed algorithm solving the problem.

The basic idea in their algorithm is to transfer the privilege for entering the critical sections with a single privilege message. Figure 1 (b) shows the algorithm for node $I \in \{1,2,\ldots,N\}$ in a traditional style. *HavePrivilege* and *Requesting* are boolean variables indicating whether node I owns the privilege and wants to enter or stays in its critical section, respectively. Q is a queue of integers, and *RN* and *LN* are integer arrays of size N. Q holds IDs of nodes that wait to enter their critical sections if node I owns the privilege. $LN[j]$ for $j \in \{1,2,\ldots,N\}$ is the sequence number of the node j's request granted most recently if node I owns the privilege. *RN* records the largest request number



(a) transition rules modeling Suzuki&Kasami algorithm

(b) Suzuki&Kasami distributed mutual exclusion algorithm

Figure 1. Correspondence between transition rules modeling Suzuki&Kasami algorithm and statements in Suzuki&Kasami algorithm

ever received from each one of the other nodes. Node I uses $RN[I]$ to generate its own sequence numbers. Initially, *HavePrivilege* is true in node 1 and false in any other node, *Requesting* is false in all nodes, Q is empty in all nodes, and $RN[i]$ and $LN[i]$ for $i \in \{1, 2, \ldots, N\}$ are zero in all nodes.

If node i wants to enter its critical section, it first calls its own procedure *P1*. If it happens to own the privilege, it immediately enters the critical section. Otherwise, it generates the next sequence number, that is, incrementing $RN[i]$, and sends REQUEST$(i, RN[i])$ to all other nodes. If it receives a privilege message, it enters the critical section. When it finishes executing the critical section, it sets $LN[i]$ to its current sequence number $RN[i]$, indicating that the current request has been granted, and updates Q, namely that IDs of nodes that want to enter their critical sections and are not in Q yet are added to

Q. After that, if Q is not empty, node i sets *HavePrivilege* to false and sends PRIVILEGE($tail(Q), LN$) to the node found in the front of Q, and otherwise, node i keeps the privilege. Finally node i sets *Requesting* to false and leaves procedure *P1*.

Whenever REQUEST(j, n) is delivered to node i, node i can execute its own procedure *P2*. But, procedure *P2* has to be atomically executed. When node i executes procedure *P2*, it sets $RN[j]$ to n if n is greater than $RN[j]$. Then, if node i owns the privilege, neither wants to enter nor stays in its critical section, and the nth request of node j has not been granted, that is, $RN[j] = LN[j]+1$, then it sets *HavePrivilege* to false and sends PRIVILEGE(Q, LN) to node j.

5. Modeling a Computer Network

In the underlying computer network, as described, the communication delay is totally unpredictable. There is no assumption about network topology. Such a computer network is modeled as OTS $\mathcal{S}_M$ with a set of variables $\{msgs_i \mid i \in \mathbf{N}^+\}$ and two sets of transition rules $\{put_{\{m,i\}} \mid m \in Msg \wedge i \in \mathbf{N}^+\}$ and $\{del_{\{m,i\}} \mid m \in Msg \wedge i \in \mathbf{N}^+\}$, where $\mathbf{N}^+$ is a set of positive integers and *Msg* is a set of messages. When $\mathcal{S}_M$ is used to model Suzuki&Kasami algorithm, $\mathbf{N}^+$ is restricted to $\{1, 2, \ldots, N\}$ and *Msg* is instantiated as $\{\text{PRIVILEGE}(q, a) \mid q \in Queue(\{1, 2, \ldots, N\}) \wedge a \in Array[N](\mathbf{N})\} \cup \{\text{REQUEST}(i, n) \mid 1 \leq i \leq N \wedge n \in \mathbf{N}^+\}$, where $Queue(\{1, 2, \ldots, N\})$ is a set of queues of positive integers which range is $\{1, 2, \ldots, N\}$ and $Array[N](\mathbf{N})$ is a set of natural number arrays of size N.

The value of variable $msgs_i$ is a multiset, or a bag of messages addressed to node i, and is initially empty. A bag of messages is used to represent the character of the computer network, namely that the communication delay is totally unpredictable. Transition rules $put_{\{m,i\}}$ and $del_{\{m,i\}}$ denote that message m is sent to node i and message m addressed to node i is deleted, respectively. $put_{\{m,i\}}$ is always effective, and $del_{\{m,i\}}$ is effective if and only if (iff) $msgs_i$ holds message m. If $put_{\{m,i\}}$ is executed, message m is put to $msgs_i$, and if $del_{\{m,i\}}$ is executed in a state in which $msgs_i$ holds m, m is deleted from $msgs_i$.

$\mathcal{S}_M$ is described in CafeOBJ. Basic units of CafeOBJ specifications are modules that may have parameters. $\mathcal{S}_M$ is written as one module called `MEDIUM` with one parameter that specifies the sort of messages exchanged by nodes. Moreover, a module may import other modules so as to use sorts and operations declared in the modules. Module `MEDIUM` imports two other modules `NAT` and `BAG` in which sorts and operations related to natural numbers and bags are declared. `BAG` is a parameterized module with one parameter. The state space of $\mathcal{S}_M$ is represented by hidden sort `Medium`, a set of vari-

ables $\{msgs_i \mid i \in \mathbf{N}^+\}$ by observation `msgs`, and two sets of transition rules $\{put_{\{m,i\}} \mid m \in Msg \wedge i \in \mathbf{N}^+\}$ and $\{del_{\{m,i\}} \mid m \in Msg \wedge i \in \mathbf{N}^+\}$ by actions `put` and `del` respectively.

The following is the specification of $\mathcal{S}_M$ in CafeOBJ:

```
mod* MEDIUM (D :: TRIV) { -- parameter D constrained by module TRIV.
  pr(NAT + BAG(D))        -- imports two modules; BAG instantiated with D.
  *[Medium]*
-- any initial state
  op imed  :                    -> Medium -- imed is any initial state of S_M.
-- observations
  bop msgs : Medium NzNat       -> Bag -- NzNat stands for positive integers.
-- actions
  bop put  : Medium Elt.D NzNat -> Medium
  bop del  : Medium Elt.D NzNat -> Medium
-- CafeOBJ variables
  var  M   : Medium
  vars I J : NzNat
  var  X   : Elt.D
-- in any initial state
  eq  msgs(imed,I)       = void . -- that is, empty.
-- after execution of 'put'
  eq  msgs(put(M,X,I),I) = X,msgs(M,I) . -- comma ',' is a constructor for Bag.
  ceq msgs(put(M,X,J),I) = msgs(M,I) if I =/= J .
-- after execution of 'del'
  eq  msgs(del(M,X,I),I) = msgs(M,I) - X . -- minus '-' is a bag difference.
  ceq msgs(del(M,X,J),I) = msgs(M,I) if I =/= J .
}
```

A comment starts with '-' and terminates at the end of the line. An actual parameter of module `MEDIUM` is constrained by module `TRIV` in which one visible sort `Elt` is declared, that is, the actual parameter has to have at least one visible sort. When `MEDIUM` is instantiated with an actual parameter, visible sort `Elt.D` is replaced with some visible sort declared in the parameter such as `Msg`.

Given a state s of `Medium` and a positive integer i, $\texttt{msgs}(s,i)$ denotes the value of variable $msgs_i$, and besides given a message x, $\texttt{put}(s,x,i)$ (or $\texttt{del}(s,x,i)$) denotes the state after executing transition rule $put_{\{x,i\}}$ (or $del_{\{x,i\}}$) in state s. Equations are used to specify any initial state of $\mathcal{S}_M$ and to define what happens after executing each transition rule.

6. Modeling Suzuki&Kasami Algorithm

Suzuki&Kasami algorithm is modeled as OTS $\mathcal{S}_{SK}$ with seven sets of variables, 13 sets of transition rules and one sub-system. The 13 sets of transition rules are $\{try_i\}$, $\{setReq_i\}$, $\{prv?_i\}$, $\{genSN_i\}$, $\{mcREQ_i\}$, $\{wtPRV_{\{m,i\}}\}$, $\{exit_i\}$, $\{endReq_i\}$, $\{updQ_i\}$, $\{empty?_i\}$, $\{sdPRV_i\}$, $\{ustReq_i\}$ and $\{rcREQ_{\{m,i\}}\}$, where $1 \leq i \leq N$ and m is any privilege or request message. Figure 1 shows which transition rule corresponds to which statements in Suzuki&Kasami algorithm. The seven sets of variables are $\{loc_i\}$, $\{prv_i\}$, $\{req_i\}$, $\{q_i\}$, $\{rn_i\}$, $\{ln_i\}$, and $\{iter_i\}$, where $1 \leq i \leq N$. loc_i indicates the location at which node i is, where locations are *rem*, *l1*, *l2*, etc. shown in Fig. 1 (a). prv_i, req_i, q_i, rn_i, ln_i, and $iter_i$ correspond to

HavePrivilege, *Requesting*, Q, *LN*, *RN*, and j in Suzuki&Kasami algorithm shown in Fig. 1 (b), respectively. The one sub-system is $\mathcal{S}_M$ that represents the underlying computer network. Initially, every loc_i and $iter_i$ are *rem* and 0 respectively, every $msgs_i$ of $\mathcal{S}_M$ is empty, and the rest are the same as the initial values of the corresponding variables in Suzuki&Kasami algorithm.

The following show 1) the condition on which each transition rule is effective, and 2) how each variable changes if each transition rule is executed in a state in which it is effective, but do not show variables that are left unchanged:

try_i (1) $loc_i = rem$,
(2) $loc_i := l1$.

$setReq_i$ (1) $loc_i = l1$,
(2) $req_i := true$; $loc_i := l2$.

$prv?_i$ (1) $loc_i = l2$,
(2) **if** $prv_i = true$ **then** $loc_i := cs$ **else** $loc_i := l3$.

$genSN_i$ (1) $loc_i = l3$,
(2) $rn_i[i] := rn_i[i] + 1$; $iter_i := 1$; $loc_i := l4$.

$mcREQ_i$ (1) $loc_i = l4$,
(2) **if** $iter_i \leq N$ **then**
(**if** $iter_i \neq i$ **then** $\mathcal{S}_M.put_{\{\text{REQUEST}(i,rn_i[i]),iter_i\}}$);
$iter_i := iter_i + 1$ **else** $loc_i := l5$.

$wtPRV_{\{m,i\}}$ (1) $loc_i = l5 \wedge m = \text{PRIVILEGE}(Q, LN) \wedge m \in \mathcal{S}_M.msgs_i$,
(2) $\mathcal{S}_M.del_{\{m,i\}}$; $prv_i := true$; $q_i := Q$; $ln_i := LN$; $loc_i := cs$.

$exit_i$ (1) $loc_i = cs$,
(2) $loc_i := l6$.

$endReq_i$ (1) $loc_i = l6$,
(2) $ln_i[i] := rn_i[i]$; $iter_i := 1$; $loc_i := l7$.

$updQ_i$ (1) $loc_i = l7$,
(2) **if** $iter_i \leq N$ **then**
(**if** $iter_i \neq i \wedge iter_i \in q_i \wedge rn_i[iter_i] = ln_i[iter_i] + 1$ **then**
$put(q_i, iter_i)$); $iter_i := iter_i + 1$ **else** $loc_i := l8$.

$empty?_i$ (1) $loc_i = l8$,
(2) **if** $q_i = void$ **then** $loc_i := l10$ **else** $loc_i := l9$.

$sdPRV_i$ (1) $loc_i = l9$,
(2) $prv_i := false$; $\mathcal{S}_M.put_{\{\text{PRIVILEGE}(tail(q_i),ln_i),head(q_i)\}}$;
$loc_i := l10$.

$ustReq_i$ (1) $loc_i = l10$,
(2) $req_i := false$; $loc_i := rem$.

$rcREQ_{\{m,i\}}$ (1) $m = \text{REQUEST}(j, n) \wedge m \in \mathcal{S}_M.msgs_i$,
(2) $\mathcal{S}_M.del_{\{m,i\}}$; $rn_i[j] := max(rn_i[j], n)$;
if $prv_i = true \wedge req_i = false \wedge rn_i[j] = ln_i[j] + 1$ **then**
$prv_i := false$; $\mathcal{S}_M.put_{\{\text{PRIVILEGE}(q_i,ln_i),j\}}$.

A variable x and a transition rule τ defined in $\mathcal{S}_M$ are represented by $\mathcal{S}_M.x$ and $\mathcal{S}_M.\tau$ respectively in the above.

$\mathcal{S}_{SK}$ is then described in CafeOBJ using module `MEDIUM`. `MEDIUM` is instantiated with module `MSG` in which sorts and operations related to privilege and request messages are declared, and `Elt.D` is replaced with `Msg` denoting such messages. The main part of the signature is as follows:

```
    *[Sys]*                              -- Sys for the state space of S_SK.
  -- any initial state
    op   init     : -> Sys               -- init denotes any initial state.
  -- observations
    bop  loc      : Sys NzNat -> Loc     -- Loc for locations, e.g. rem and l1.
    bops prv req  : Sys NzNat -> Bool    -- Bool for boolean values.
    bop  q        : Sys NzNat -> Queue   -- Queue for queues of positives.
    bops rn ln    : Sys NzNat -> Array   -- Array for integer arrays.
    bop  iter     : Sys NzNat -> NzNat
  -- actions
    bops try setReq prv? genSN mcREQ exit : Sys NzNat     -> Sys
    bops endReq updQ empty? sdPRV ustReq  : Sys NzNat     -> Sys
    bops wtPRV rcREQ                      : Sys Msg NzNat -> Sys
  -- projections
    bop  med                              : Sys           -> Medium
```

Projection `med` is used to denote sub-system $\mathcal{S}_M$.

We have basically 14 sets of equations in the specification: one for any initial state, and the others for the 13 actions. In this paper, we present one set of equations for action `wtPRV`. The equations defining how a state of $\mathcal{S}_{SK}$ changes if transition rule $wtPRV_{\{M,I\}}$ is executed in the state are as follows:

```
ceq loc(wtPRV(S,M,I),I)  = cs
    if loc(S,I) == l5 and M \in msgs(med(S),I) and p?(M) .
ceq loc(wtPRV(S,M,I),J)  = loc(S,J)
    if I =/= J or loc(S,I) =/= l5 or not(M \in msgs(med(S),I)) or not(p?(M)) .
ceq prv(wtPRV(S,M,I),I)  = true
    if loc(S,I) == l5 and M \in msgs(med(S),I) and p?(M) .
ceq prv(wtPRV(S,M,I),J)  = prv(S,J)
    if I =/= J or loc(S,I) =/= l5 or not(M \in msgs(med(S),I)) or not(p?(M)) .
eq  req(wtPRV(S,M,I),J)  = req(S,J) .
ceq q(wtPRV(S,M,I),I)    = getQ(M)
    if loc(S,I) == l5 and M \in msgs(med(S),I) and p?(M) .
ceq q(wtPRV(S,M,I),J)    = q(S,J)
    if I =/= J or loc(S,I) =/= l5 or not(M \in msgs(med(S),I)) or not(p?(M)) .
eq  rn(wtPRV(S,M,I),J)   = rn(S,J) .
ceq ln(wtPRV(S,M,I),I)   = getA(M)
    if loc(S,I) == l5 and M \in msgs(med(S),I) and p?(M) .
ceq ln(wtPRV(S,M,I),J)   = ln(S,J)
    if I =/= J or loc(S,I) =/= l5 or not(M \in msgs(med(S),I)) or not(p?(M)) .
eq  iter(wtPRV(S,M,I),J) = iter(S,J) .
ceq med(wtPRV(S,M,I))    = del(med(S),M,I)
    if loc(S,I) == l5 and M \in msgs(med(S),I) and p?(M) .
ceq med(wtPRV(S,M,I))    = med(S)
    if loc(S,I) =/= l5 or not(M \in msgs(med(S),I)) or not(p?(M)) .
```

`S`, `M`, `I` and `J` are CafeOBJ variable whose sorts are `Sys`, `Msg`, `NzNat` and `NzNat`, respectively. `p?` is a predicate examining whether its argument is a privilege message, and `getQ` and `getA` extract the queue and array from a privilege message.

7. Verification with CafeOBJ System

There are several levels in using CafeOBJ system as an interactive proof-checker or verifier. The followings are three typical ones:

- **Proof Assistant**: You write mathematical proofs in natural language based on CafeOBJ specifications, and have CafeOBJ system assist necessary logical inferences and/or calculations.
- **Proof Score Executor**: You write proof scores in CafeOBJ, and have CafeOBJ system execute the proof scores. If the results are as expected, you can be confident that the specified systems have properties at issue.
- **Automatic Verifier or Model Checker**: You write assertions that should be proved, and have CafeOBJ system verify the assertions automatically. PigNose resolution engine of CafeOBJ system is usually used, and you are supposed to set several parameters of the engine appropriately.

Each of the three levels has its own merit and demerit, and should be selected depending on problems to be solved and situations.

8. Verification of Suzuki&Kasami Algorithm

We have verified that Suzuki&Kasami algorithm has one safety property and one liveness property based on the specification using UNITY logic with CafeOBJ system partly as a proof assistant and partly as a proof score executor. The safety property is *mutual exclusion*, and the liveness one is *lockout freedom*.

Mutual exclusion. Two sub-claims are needed to prove the main claim.

Claim 1 *In any reachable state, if for any node* i, $loc_i = cs$, *l6, l7, l8, or l9, then* $prv_i = true$.

Proof This claim is proved by induction on transition rules. Since every node is initially at location *rem*, the claim is vacuously true in any initial state. Thus, it suffices that the claim is shown to be preserved by every transition rule. It is straightforward to show that every transition rule preserves the claim. □

Claim 2 *In any reachable state, there exists either one and only node that owns the privilege or one and only privilege message in the network.*

Proof This claim is proved by induction on transition rules. Since initially only node 1 owns the privilege and there is no privilege message, the claim is clearly true in any initial state. Thus, it suffices that the claim is shown

to be preserved by every transition rule. Suppose that the claim holds in a state s, we show that it still holds in the successor state s' after executing any transition rule in s. For all transition rules except $wtPRV_{\{m,i\}}$, $sdPRV_i$, and $rcREQ_{\{m,i\}}$, it is straightforward to show this. In this paper, we only show the proof that $wtPRV_{\{m,i\}}$ preserves the claim. We can prove that $sdPRV_i$ and $rcREQ_{\{m,i\}}$ also preserve the claim in a similar way, although Claim 1 is needed for $sdPRV_i$.

It is sufficient to consider a state in which $wtPRV_{\{m,i\}}$ is effective because its execution changes nothing if it is not effective. The condition on which it is effective is that $loc_i = l5$, m is a privilege message and $m \in \mathcal{S}_M.msgs_i$. From the hypothesis, there does not exist a node that owns the privilege nor any other privilege message except m in such a state. All we have to do is to show that only node i owns the privilege and there exists no privilege message in s'. The following proof score can be used to show this:

```
open AlgorithmA
  ops s s' : -> Sys .   -- s denotes any assumed state.
  ops i j  : -> NzNat . -- j denotes any other node except i.
  op  m    : -> Msg .   -- m is one and only privilege message in med(s).
  op  b    : -> Bag .
  eq  loc(s,i)       = l5 .   -- from the assumption.
  eq  msgs(med(s),i) = m,b .  -- from the assumption.
  eq  p?(m)          = true . -- it states that m is a privilege message.
  eq  s'             = wtPRV(s,m,i) . -- s' is the successor state.
  red     prv(s',i) == true    and prv(s',j) == prv(s,j)
      and msgs(med(s'),i) == b and msgs(med(s'),j) == msgs(med(s),j) .
close
```

`AlgorithmA` is a module in which the specification of Kasami&Suzuki algorithm is written. By opening the module using CafeOBJ command `open`, the definitions in the specification can be used. The expression following CafeOBJ command `red(uction)` means that only node i owns the privilege and there exists no privilege message in s'. `red` reduces the expression by regarding equations as left-to-right rewrite rules. In this case, the expression is reduced to `true`. □

Claim 3 (mutual exclusion) *In any reachable state, there is at most one node which is at location cs, l6, l7, l8, or l9.*

Proof The claim immediately follows from Claims 1 and 2. □

Lockout freedom. Suppose that every node *repeatedly* tries to enter its critical section, we show that any node that wants to enter its critical section eventually enters there. 15 sub-claims are needed to show the main claim, 13 of which (including Claims 1 and 2) relate to safety properties and two of which to liveness ones. In this paper, we only present the sub-claims explicitly needed for the proof of the main claim.

We first define *gq* and *gln* as follows:

Definition 1 *From Claim 2, gq and gln can be defined to be total functions of states of* $\mathcal{S}_{SK}$.

$$(gq, gln) = \begin{cases} (q_i, ln_i) & \textbf{if } \textit{there exists node } i \textit{ such that } prv_i = true \\ (Q, LN) & \textbf{if } \textit{there exists } \mathrm{PRIVILEGE}(Q, LN). \end{cases}$$

Claim 4 *In any reachable state, if for any node i,* $loc_i = l4$ *or* $l5$*, then* $rn_i[i] = gln[i] + 1$.

Claim 5 *In any reachable state, if for any node i, there exists a privilege message addressed to node i, then* $loc_i = l4$ *or* $l5$.

Claim 6 *In any reachable state, if for any nodes i, a request message is sent to node i, node i eventually receives it.*

Claim 7 *In any reachable state, if for any node i, a privilege message is sent to node i, node i eventually receives it.*

Claim 8 *In any reachable state, if for any node i and any positive integer n, a node receives* $\mathrm{REQUEST}(i, n)$*, then* $n \leq gln[i] + 1$.

Claim 9 *In any reachable state,* $rn_j[i] \leq gln[i] + 1$ *for any nodes* i, j *but* $i \neq j$.

Claim 10 (lockout freedom) *A node that wants to enter the critical section eventually enters there. More formally, for any node i,* $loc_i = l1 \mapsto loc_i = cs$.

Proof It is straightforward to show $loc_i = l1 \mapsto loc_i = l2$, $loc_i = l2 \wedge prv_i = true \mapsto loc_i = cs$, and $loc_i = l2 \wedge prv_i = false \mapsto loc_i = l5$. Thus, all that is needed is to show $loc_i = l5 \mapsto loc_i = cs$.

While node i is at $l4$, it sends $\mathrm{REQUEST}(i, n)$ where $n = rn_i[i]$ to any other node j. At this moment, from Claim 4, $rn_i[i] = gln[i]+1$. From Claim 6, any other node j eventually receives $\mathrm{REQUEST}(i, n)$. When node j receives $\mathrm{REQUEST}(i, n)$, from Claim 8, $n \leq gln[i] + 1$. If $n < gln[i] + 1$ at this moment, node i must have already entered its critical section for the request at issue because $gln[i]$ can be modified iff $endReq_i$ is executed when $loc_i = l6$. Therefore, it is sufficient to consider the case in which when each node j $(\neq i)$ receives $\mathrm{REQUEST}(i, n)$, $n = gln[i] + 1$. If $prv_j = true$, $req_j = false$, and $max(rn_j[i], n) = ln_j[i]+1$ when node j receives $\mathrm{REQUEST}(i, n)$, node j sends a privilege message to node i, after which node i eventually receives the message from Claim 7 and enters its critical section. Thus, moreover, we suppose that when each node j $(\neq i)$ receives $\mathrm{REQUEST}(i, n)$, $prv_j = false$, $req_j = true$, or $max(rn_j[i], n) \neq ln_j[i] + 1$.

In the supposed case, if each node j $(\neq i)$ receives $\mathrm{REQUEST}(i, n)$, then $rn_j[i] = gln[i] + 1$ from Claim 9. Hereafter, unless node i enters its critical

section, the equation $rn_j[i] = gln[i] + 1$ keeps holding from Claim 9 because $gln[i]$ can be modified iff $endReq_i$ is executed when $loc_i = l6$, and $rn_j[i]$ never decreases.

Since every node *repeatedly* tries to enter its critical section, from Claims 2 and 5, some node j eventually gets to $l7$ where $iter_j = i$ after all nodes except node i have received REQUEST(i, n). At this moment, $rn_j[i] = gln[i] + 1$ from the above. Thus, node j puts i into gq or i has been already in gq.

From what we have discussed so far, $loc_i = l4 \mapsto i \in gq$.

Let us define a partial function $d : Queue\ NzNat \rightarrow NzNat$ as follows: $d(q, i) = \mathbf{if}\ hd(q) = i\ \mathbf{then}\ 1\ \mathbf{else}\ d(tl(q)) + 1$.

From now on, we show the following:

$$loc_i \in \{l4, l5\} \wedge d(gq, i) = k \mapsto (loc_i \in \{l4, l5\} \wedge d(gq, i) < k) \vee loc_i = cs\ . \tag{1}$$

From Claims 2 and 5, some node j eventually gets to $l9$ because gq is not empty, and effectively executes $sdPRV_i$. The case is divided into two subcases: i) $k = 1$, and ii) $k > 1$. In case (i), node j sends a privilege message to node i, after which node i eventually receives the message from Claim 7 and eventually enters its critical section. In case (ii), gq becomes $tl(gq)$. That is, $d(tl(gq), i) = k - 1 < k$.

Applying Induction theorem for leads-to (Chandy and Misra, 1988) to (1), we obtain the following: $loc_i \in \{l4, l5\} \mapsto loc_i = cs$. □

9. Concluding Remarks

In the verification that Suzuki&Kasami algorithm is lockout free, we have used the assumption that each node *repeatedly* tries to enter its critical section. If this assumption is not used, there is a counter example indicating that the algorithm may cause lockout. Suppose that a computer network consists of two nodes and that node 1 executes its critical section only once. Let us consider the following case. Node 1 finishes executing its critical section and is at location *l10*. While node 1 stays in *l10*, node 2 executes *P1* to enter its critical section and sends a request message to node 1. After that, node 1 receives the request message, executing *P2*, and then node 1 sets *Requesting* to false and leaves *P1*. Node 2 will continue forever waiting to receive a privilege message from node 1 because node 1 never executes *P1* more than once, never transferring the privilege to node 2.

We found this counter example when we were carefully considering how to prove the algorithm lockout free. Although we might have found such a counter example without formal methods, we believe that formal methods are useful tools making it possible for us to deeply understand problems through modeling the problems and/or verifying that they have some properties, which reduces errors.

References

CafeOBJ web page (2001). CafeOBJ web page. http://www.ldl.jaist.ac.jp/cafeobj/.

Chandy, K. M. and Misra, J. (1988). *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA.

Diaconescu, R. and Futatsugi, K. (1998). *CafeOBJ Report*. AMAST Series in Computing, 6. World Scientific, Singapore.

Diaconescu, R., Futatsugi, K., and Iida, S. (1999). Component-based algebraic specification and verification in CafeOBJ. In *Formal Methods '99 Conference Proceedings (LNCS 1709, Springer-Verlag)*, pages 1644–1663.

Goguen, J. and Malcolm, G. (2000). A hidden agenda. *TCS*, 245:55–101.

Lamport, L. (1994). The temporal logic of actions. *ACM TOPLAS*, 16(3):872–923.

Lynch, N. A. (1996). *Distributed algorithms*. Morgan-Kaufmann, San Francisco, CA.

Manna, Z. and Pnueli, A. (1991). *The temporal logic of reactive and concurrent systems: specification*. Springer-Verlag, NY.

Manna, Z. and Pnueli, A. (1995). *Temporal verification of reactive systems: safety*. Springer-Verlag, NY.

Suzuki, I. and Kasami, T. (1985). A distributed mutual exclusion algorithm. *ACM TOCS*, 3(4):344–349.